

---

# Lektion 3

---

In Ihren Programmiervorlesungen haben Sie das Prinzip der Rekursion bereits kennengelernt. Typischerweise werden dort zur Einführung nur sehr einfache Anwendungsbeispiele vorgestellt, die auch leicht hätten ohne Rekursion gelöst werden können. In diesem Kapitel werden Sie eine Reihe weitergehende, nichttriviale Anwendungen der Rekursion kennenlernen, die mit rein iterativen (nur auf Schleifen basierenden) Algorithmen nur sehr schwer zu lösen wären. Ein wichtiger Punkt wird auch sein, wie die Korrektheit rekursiver Algorithmen bewiesen werden kann.

## Kap.3 Rekursive Algorithmen

---

### *Inhalt*

- ▶ Prinzip der Rekursion
- ▶ Wie kommt man zu rekursiven Problemlösungen?
- ▶ Korrektheit rekursiver Algorithmen: Wie kann man sich überlegen, ob ein rekursives Programm korrekt arbeitet?
- ▶ Anwendungsbeispiel Kombinatorik: Menge aller Teilmengen berechnen
- ▶ Rekursive Suche nach Lösungen: Algorithmisches Prinzip "Backtracking"

### 3.1 Prinzip der Rekursion

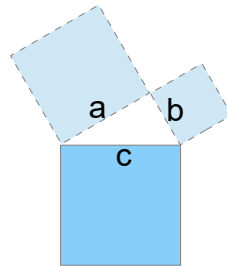
---

Rekursion gibt es in vielfältigen Formen, nicht nur beim Programmieren. Hier ist ein Beispiel in grafischer Form, ein sog. Pythagorasbaum.

#### *Beispiel 3.1 - Pythagorasbaum*



Grundlage ist die Darstellung der Formel  $a^2 + b^2 = c^2$  nach Satz des Pythagoras.



Der Ausgangspunkt ist dabei ein Quadrat der Seitenlänge c, auf das ein rechtwinkliges Dreieck gesetzt wird. Auf den Seiten der kleineren Teilquadrate jeweils gegenüberliegend zu den Seiten a und b des Dreiecks wird dann in gleicher Weite (d.h. rekursiv) weiter gezeichnet, bis eine angegebene Rekursionstiefe erreicht wird.

### *Rekursiver Lösungsansatz für Problemstellungen*

Aus Sicht programmiersprachlicher Konzepte ist Rekursion eigentlich sehr einfach: Eine Methode darf sich (direkt oder indirekt über andere Methoden) selbst wieder aufrufen. Das Interessante – und auch schwierige – an Rekursion ist aber, dass damit eine andere Denkweise und Herangehensweise an Probleme verbunden ist als bei den gewohnten iterativen (d.h. schleifenbasierten) Lösungen.

### *Vorgehensweise 3.2 - Entwurf rekursiver Problemlösungen*

1. Sofern die Problemistanz so einfach ist, dass die Lösung direkt bestimmt werden kann, berechne die Lösung entsprechend.
2. Sofern die gegebene Problemistanz nicht so einfach ist, dass sie direkt lösbar ist:
  - führe die Lösung des Gesamtproblems auf die Lösung eines oder mehrerer gleichartiger, typischerweise kleinerer/einfacherer Teilproblem zurück.
  - berechne die Lösungen der Teilprobleme in gleicher Weise (d.h. rekursiv)
  - kombiniere die Lösungen der Teilprobleme zur Lösung des Gesamtproblems.

### *Beispiel 3.3 - Rekursive Suche nach Zugverbindungen*

- ▶ Ein Beispiel für rekursives Vorgehen ist die Verfahrensweise für die Suche nach einer passenden Zugverbindung von A nach B (wenn Sie keine Bahn-App, sondern nur gedruckte Fahrpläne zur Verfügung hätten):
  - a) Gibt es eine zeitlich passende direkte Verbindung von A nach B, nehme diese Verbindung
  - b) Gibt es keine passende direkte Verbindung von A nach B, dann
    1. wähle einen Umsteigebahnhof C zwischen A und B,

2. bestimme **Verbindung von A nach C** (rekursiv in gleicher Weise) und
3. bestimme (Anschluss-) **Verbindung von C nach B** (rekursiv in gleicher Weise).

Die Gesamtverbindung setzt sich zusammen aus der Verbindung von A nach C und der Anschlussverbindung von C nach B.

Nun ein einfaches Beispiel im Programmierumfeld:

### 3.1.1 Beispiel Summe eines Arrays

Es soll die Summe aller Werte eines Arrays rekursiv (ohne Schleifen) berechnet werden.

```
public static double sum(double[] arr)
```

Bei Bedarf können auch Hilfsmethoden eingeführt werden.

Um eine rekursive Lösung zu finden, sind folgende zwei Fragen zu klären:

- a) In welchen Fällen kann die Lösung direkt angegeben werden?

Sofern der zu summierende Bereich nur ein Element umfasst, ist dieses Element auch die Summe.

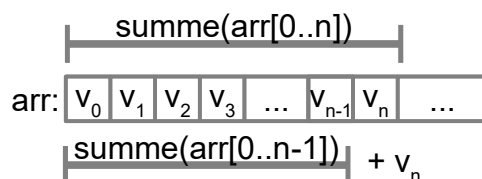
- b) Wie kann das Problem, einen Bereich aus  $n$  Elementen zu summieren (für  $n > 1$ ), auf ein oder mehrere gleichartige "kleinere" Probleme zurückgeführt werden?

Dazu werden nachfolgend zwei verschiedene Versionen betrachtet.

Oft ist es dabei rekursiven Lösungen so, dass man nicht direkt das gegebene Problem nehmen kann, sondern ein etwas verallgemeinertes Problem betrachten muss. Das ist auch hier der Fall. Wir betrachten als verallgemeinertes Problem, dass nicht ein komplettes Array `arr` zu summieren ist, sondern ein Teilbereich `arr[0..m]` eines Arrays. Wenn wir eine Lösung dafür haben, dann können wir damit natürlich auch das gesamte Array `arr[0 .. arr.length-1]` summieren.

#### Rekursive Summenberechnung – Version 1 (lineare Rekursion)

Ein naheliegender Ansatz ist der, die Summe von  $n$  Werten auf die Summe von  $n-1$  Werten zurückzuführen. Um die Summe von  $n$  Werten zu bilden, berechnet man erst (rekursiv) die Summe von  $n-1$  Werten und addiert dann den  $n$ -ten Wert dazu.

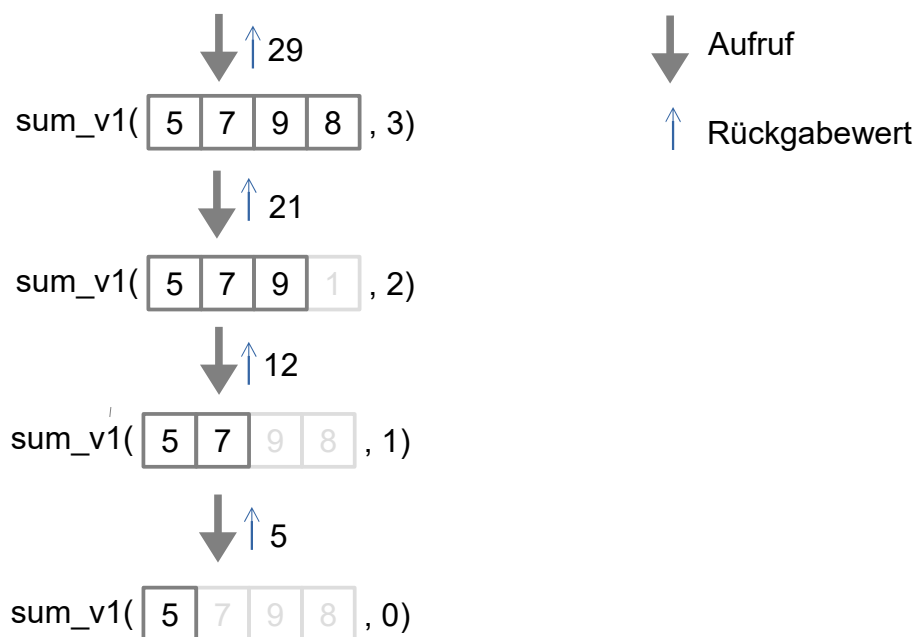


Dies ist in Java einfach umzusetzen und ergibt folgende **linear rekursive** Lösung. **Linear rekursiv** bedeutet, dass jeder Aufruf zu höchstens einem weiteren Aufruf führt.

```
public static double sum_v1(double[] arr) {
    return sum_v1(arr, arr.length-1);
}

/** Berechnet Summe der Werte des Teilarrays arr[0..lastIndex] */
private static double sum_v1(double[] arr, int lastIndex) {
    if (lastIndex == 0) {
        return arr[0];
    } else {
        return sum_v1(arr, lastIndex-1) + arr[lastIndex];
    }
}
```

Hier ist ein Beispiel für den Ablauf einer Berechnung schematisch dargestellt:



Die lineare Aufrufreihenfolge, die sich ergibt, ist gut zu erkennen.

### Aufgabe 3.4 - sum\_v1 testen

In Moodle finden Sie die Methode `sum_v1`. Testen Sie die Methode mit Arrays unterschiedlicher Länge (z.B. mit Arrays der Längen  $n = 1000$ ,  $10\,000$ ,  $100\,000$ ,  $1\,000\,000$ ), ob die Methode richtig funktioniert.

### Beobachtung

Wenn Sie die Tests durchgeführt haben, haben Sie vermutlich festgestellt, dass die

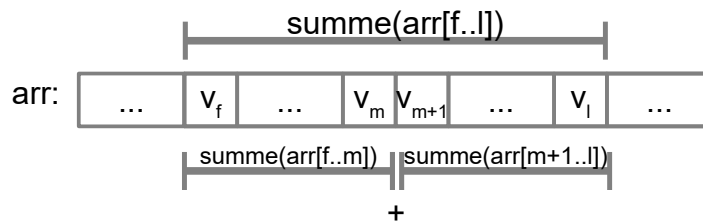
Methode im Prinzip korrekt funktioniert (was auch nicht verwunderlich ist), dass es aber bei langen Feldern ein Problem gibt - die Methode führt zu einem `StackOverflow-Error`.

Warum? Sie haben in Programmieren gelernt, dass für jeden Methodenaufruf ein sog. Stackframe für die lokalen Daten und Verwaltungsdaten einer Methode auf den Aufrufstack (einen Teil der Speicherverwaltung) gelegt wird. Der Aufrufstack enthält alle noch aktiven, d.h. noch nicht beendeten Methodenaufrufe. Durch die linear rekursive Lösung bekommt man bei einem Array der Länge  $n$  dann auch bis zu  $n$  aktive Aufrufe. Bei großem  $n$  führt das dazu, dass der begrenzte Speicherplatz für den Aufrufstack nicht mehr ausreicht.

Gibt es eine rekursive Lösung, die dieses Stack-Overflow-Problem vermeidet, so dass die Berechnung auch für sehr lange Arrays ( $> 1\,000\,000$ ) problemlos funktioniert?

### Rekursive Summenberechnung – Version 2 (Baumrekursion)

Das Problem,  $n$  Elemente zu summieren, kann auch auf andere Weise auf kleinere, gleichartige Probleme zurückgeführt werden: Man teilt den Bereich der  $n$  Elemente in zwei (weitgehend) gleich große Hälften. Jede dieser Hälften wird dann rekursiv summiert und die beiden Teilsummen sind dann nur noch zu addieren.



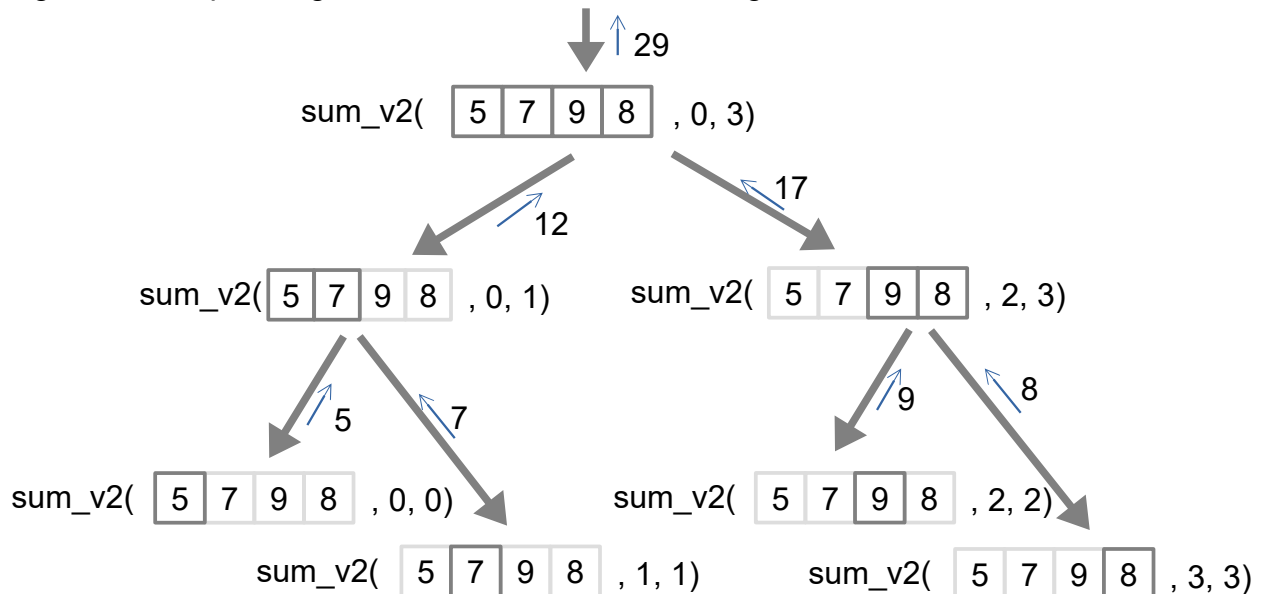
Damit ergibt sich folgende **baumrekursive Lösung**:

```
public static double sum_v2(double[] arr) {
    return sum_v2(arr, 0, arr.length-1);
}

/** Berechnet Summe der Werte von arr[firstIndex..lastIndex] */
private static double sum_v2(double[] arr, int firstIndex,
                              int lastIndex) {
    if (firstIndex == lastIndex) {
        // zu summierender Bereich besteht nur aus einem Element
        return arr[firstIndex];
    } else {
        // zu summierenden Bereich in zwei gleich große Hälften teilen
        // und rekursiv deren Summe berechnen und dann addieren
        int mid = (firstIndex + lastIndex) / 2;
        return sum_v2(arr, firstIndex, mid)
            + sum_v2(arr, mid+1, lastIndex);
    }
}
```

```
}
}
```

Folgendes Beispiel zeigt die einzelnen Teilberechnungen



Man erkennt deutlich, warum das *Baumrekursion* genannt wird.

Ist das Stack-Overflow-Problem damit gelöst? Wie viele Aufrufe landen maximal auf dem Aufrufstack?

Die längste gleichzeitig aktive Aufruffolge auf dem Stack entspricht dem längsten Pfad von der Wurzel des Baum zu den Blättern, also der Höhe des Baums. Da die zu summierenden Bereiche jeweils halbiert werden, hat man eine maximale Rekursionstiefe von ca.  $\log_2(n)$  bei Arraylänge  $n$ . Bei  $n = 1\,000\,000$  also nur ca. 20 Aufrufe auf dem Stack statt 1 Mio., wie bei Version 1. Dies stellt für die Ausführung dann kein Problem mehr dar.

### Aufgabe 3.5 - Rekursive Minimumsbestimmung

- Programmieren Sie in ähnlicher Weise rekursiv, d.h. ohne Schleifen, eine Methode

```
public static int minIndex(double[] arr,
                           int startIndex, int endIndex)
```

die die Position des Minimums der Werte des Array `arr` im Bereich von Index `startIndex` bis Index `endIndex` bestimmt.

Mit Hilfe der Methode `minIndex` aus der vorigen Aufgabe ist es möglich, eine rekursive Version von Selectionsort (Sortieren durch Auswahl) zu implementieren. Selectionsort haben Sie vermutlich in der üblichen Version mit Schleifen schon in der Programmiervorlesung kennengelernt. Es wird dabei nach und nach eine aufsteigende

Folge gebildet, indem jeweils von den noch unsortierten Daten der kleinste Wert entnommen und an den sortierten Bereich angefügt wird.

### Aufgabe 3.6 - Rekursives Selectionsort

- a) (etwas kniffliger) Programmieren Sie rekursiv (!) eine Methode

```
public static void selectionSort(double[] arr,  
                                int startIndex, int endIndex)
```

die nach dem Prinzip "Sortieren durch Auswahl" (selection sort) die Daten des Teilarray `arr[startIndex .. endIndex]` sortiert.

- b) Ist die Methode linear rekursiv oder ergibt sich eine Baumrekursion?

## 3.1.2 Beispiel Catalan-Zahlen

Doch nun betrachten wir anspruchsvollere Probleme, bei denen eine Lösung ohne Rekursion schwierig wäre.

### Definition/Aufgabenstellung 3.7 - Catalan-Zahlen

Die **Catalan-Zahl**  $C(n)$  ist die Anzahl der Möglichkeiten, wie ein Summe mit  $n$  Additionsoperationen geklammert werden kann.

(benannt nach dem belgischen Mathematiker Eugène Charles Catalan.)

Für kleine Werte von  $n$  können wir einfach alle Möglichkeiten durchspielen:

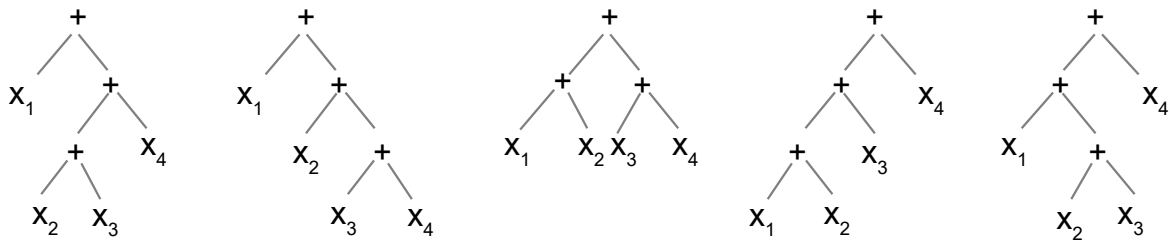
$n = 0$ : 1 Möglichkeit	$x_1$
$n = 1$ : 1 Möglichkeit	$x_1 + x_2$
$n = 2$ : 2 Möglichkeiten	$x_1 + (x_2 + x_3)$ $(x_1 + x_2) + x_3$
$n = 3$ : 5 Möglichkeiten	$x_1 + ((x_2 + x_3) + x_4)$ $x_1 + (x_2 + (x_3 + x_4))$ $(x_1 + x_2) + (x_3 + x_4)$ $(x_1 + (x_2 + x_3)) + x_4$ $((x_1 + x_2) + x_3) + x_4$

Wie kommen wir aber zu einer allgemeinen Lösung für beliebige Werte  $n$ ?

Da jeder Ausdruck als abstrakter Syntaxbaum dargestellt werden kann, könnte man die Catalan-Zahlen auch so definieren:

$C(n)$  ist die Anzahl der unterschiedlich aufgebauten abstrakten Syntaxbäume, die man mit  $n$  inneren Operator-knoten und  $(n+1)$  Blättern bilden kann.

$n = 3$ :



Wie kommt man nun zu einer Lösung? Mit einem passenden rekursiven Ansatz ist das gar nicht so schwierig.

### Rekursiver Lösungsansatz für Catalan-Zahlen

a) Für welche Fälle kann die Lösung direkt angegeben werden?

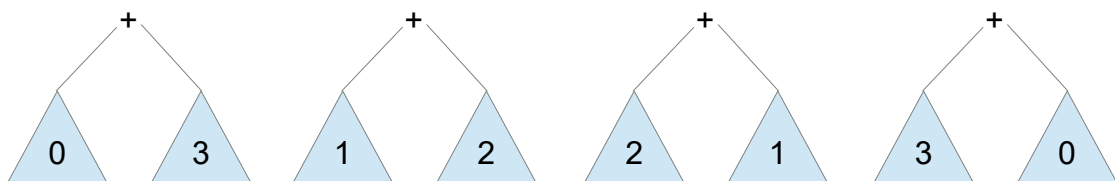
Einfachster Fall: Für  $n = 0$  gibt es nur eine Möglichkeit:  $x_1$

b) Wie kann für  $n > 0$  das Problem auf kleinere Probleme zurückgeführt werden?

Wir betrachten das Problem auf Ebene der abstrakten Syntaxbäume:

- Wähle gedanklich eine der  $n$  Additionsoperationen als Wurzel aus.
- Die restlichen  $n-1$  Operationen können dann auf den linken und den rechten Teilbaum verteilt werden.
- Wenn der linke Teilbaum  $k$  Additionsoperationen enthält, dann verbleiben für den rechten Teilbaum  $(n-1) - k$  Additionsoperationen

Beispiel: für  $n = 4$  Operatoren können im linken Teilbaum 0, 1, 2 oder 3 Operatoren enthalten sein und im rechten dann entsprechend 3, 2, 1 oder 0, so wie hier skizziert:



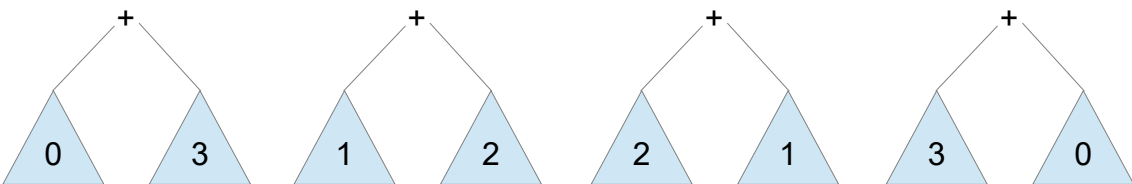
Die Zahlen geben hier die Anzahl der im Teilbaum vorkommenden Operatoren an.

- ▶ Hat der linke Teilbaum  $k$  Operatoren, dann gibt es dafür  $C(k)$  Möglichkeiten und für den rechten Teilbaum somit  $C(n-1-k)$  Möglichkeiten, wie er aufgebaut sein könnte.
  - Jeder mögliche Teilbaum links kann mit jedem möglichen Teilbaum rechts kombiniert werden. Pro Aufteilung multipliziert sich also die Anzahl der Möglichkeiten für den linken und den rechten Operanden.
- ▶ Über alle möglichen Aufteilung (wie viele Operatoren links, wie viele rechts)



muss nur noch aufsummiert werden.

Für  $n = 4$ :


$$\begin{aligned} & C(0) \cdot C(3) + C(1) \cdot C(2) + C(2) \cdot C(1) + C(3) \cdot C(0) \\ &= 1 \cdot 5 + 1 \cdot 2 + 2 \cdot 1 + 5 \cdot 1 \\ &= 14 \end{aligned}$$

Dieser Ansatz führt zu der folgenden rekursiven Summenformel, die bereits im 18ten Jahrhundert entdeckt wurde.

### Definition 3.8 - Segnersche Rekursionsformel für Catalan-Zahlen (1758)

Die Catalan-Zahl  $C(n)$  berechnet sich rekursiv nach folgender Formel:

$$C(n) = \begin{cases} 1 & \text{falls } n=0 \\ \sum_{k=0}^{n-1} C(k)C(n-1-k) & \text{falls } n>0 \end{cases}$$

Diese Formel lässt sich leicht eins-zu-eins in Java-Code umsetzen:

### Implementierung 3.9 - Catalan-Zahlen

```
public static int catalan(int n) {
    if (n == 0) {
        return 1;
    } else {
        int sum = 0;
        for (int k = 0; k < n; k++) {
            sum += catalan(k) * catalan(n-1-k);
        }
        return sum;
    }
}
```

Es handelt sich hier wieder um eine Baumrekursion, da jeder Aufruf von `catalan(n)` zu mehreren rekursiven Aufrufen auf nächster Stufe führen kann.

### Rekursion versus Iteration

In der Lösung hier wird eine for-Schleife verwendet. Die Schleife könnte zwar prinzipiell auch noch in Rekursion umgewandelt werden, was aber hier nicht wirklich sinnvoll wäre.

- ▶ Ein Ergebnis der Berechenbarkeitstheorie besagt, dass alles, was man mit Rekursion berechnen kann, sich auch ohne Rekursion (iterativ) berechnen lässt und umgekehrt.
- ▶ Für die Praxis ist es aber so, dass für manche Probleme eine rekursive Lösung naheliegend und einfach zu programmieren ist und für andere Problemstellungen eher eine iterative Lösung.
- ▶ Meine Faustregel für die Praxis ist deshalb wie folgt:  
*"Das, was mit Schleifen einfach zu machen ist, programmiere mit Schleifen.  
 Was mit Schleifen nicht einfach geht, versuche mit Rekursion."*

## 3.2 Korrektheit rekursiver Algorithmen

Rekursion wird von vielen am Anfang als schwierig empfunden, da die Berechnungsvorgänge, die durch eine rekursive Methoden, wie z.B. die `catalan`-Methode oben, hervorgerufen werden, sehr komplex sein können, so dass der gesamte Berechnungsverlauf letztendlich nicht überschaubar ist.

Wie kann man sich trotzdem überlegen, ob eine rekursive Methode, die man geschrieben hat (oder sich ausgedacht hat), richtig funktioniert? Als guter Softwareentwickler sollte man sich schon vorher überlegt haben, ob ein Ansatz funktioniert, bevor man anfängt, den Programmcode zu tippen.

Mit der passenden Denkweise und etwas Erfahrung ist das bei Rekursion relativ simpel. Es wird einfach, weil man sich gar nicht den kompletten Berechnungsvorgang vorstellen muss, sondern gewissermaßen nur eine Rekursionsstufe „weiterdenken“ muss. Die Beweismethode dafür ist die *Berechnungsinduktion*, die an folgendem Beispiel vorgestellt wird.

### Beispiel 3.10 - Potenzberechnung

Folgende rekursive Methode soll die Potenzen  $x^n$  für ganzzahlige  $n \geq 0$  berechnen.

```
public static double h(double x, int n) {
    if (n == 0) {
        return 1;
    } else if (n%2 == 0) {
        double y = h(x, n/2);
        return y * y;
    } else {
        return x * h(x, n - 1);
    }
}
```

Natürlich kann man durch Tests sich leicht davon überzeugen, dass diese einfache Methode richtig funktioniert. Wie kann man aber, ohne das Programm auszuführen, nachweisen, dass das Programm korrekt ist?

D.h. es ist zu zeigen:

$$h(x,n) = x^n \quad \text{für ganzzahliges } n \geq 0$$

Wir gehen zur Vereinfachung davon aus, dass die Zahlen und Variablen reelle bzw. natürliche Zahlen wie in der Mathematik sind und Effekte durch die endliche Zahlendarstellung (Überlaufeffekte, Rechenungenauigkeit, ...) ignoriert werden.

Wie kann man zeigen, dass eine Eigenschaft für unendlich viele mögliche Argumente gilt? Aus der Mathematik kennen Sie dafür das Beweisprinzip der vollständigen Induktion. Bei diesem Beispiel könnte man das in der Tat damit zeigen, es gibt aber ein allgemeineres Beweisverfahren, das auch bei komplizierteren rekursiven Methoden funktioniert.

### 3.2.1 Nachweis der Korrektheit rekursiv definierter Funktionen

---

Ein Beweisverfahren im Zusammenhang mit rekursiven Berechnungen ist die sog. *Berechnungsinduktion*

*Berechnungsinduktion (engl. computational induction)*

**gegeben:**

- ▶ Eine rekursiv definierte Funktion  $f(x)$ . Es wird vorausgesetzt, dass die Funktion seiteneffektfrei ist, d.h. sich wie eine Funktion im mathematischen Sinn verhält und immer für ein Argument  $x$  das gleiche Ergebnis  $y$  liefert.
- ▶ Eine Eigenschaft  $P(x, y)$ , die einen Zusammenhang zwischen dem Argument  $x$  der Funktion und dem Ergebnis  $y$  der Funktion beschreibt.

**zu beweisen:**

- ▶ Es soll gezeigt werden, dass die Eigenschaft  $P(x, f(x))$  für alle zulässigen Eingabewerte  $x$  gilt.

Die Ein- und Ausgabewerte  $x$  und  $y$  können dabei auch ganze Vektoren oder Datenstrukturen sein, die aus mehreren Werten aufgebaut sind.

*Beweismethode 3.11 - Prinzip der Berechnungsinduktion*

#### **Induktionsbasis**

- Zeige für alle zulässigen Argumente  $x$ , die nicht zu einem rekursiven Aufruf führen, dass  $P(x, f(x))$  gilt.

### Induktionsschritt(e)

- Zeige für alle zulässigen Argumente  $x$ , die zu rekursiven Aufrufen  $f(z_i)$  führen, dass die Eigenschaft  $P(x, f(x))$  gilt.
- Es kann dazu als Hypothese verwendet werden, dass  $P(z, f(z_i))$  für die rekursiven Aufrufe gilt (**Induktionshypothese**).

### Induktionsschluss:

- Dann gilt für alle zulässigen Argumente  $x$  die Eigenschaft  $P(x, f(x))$ .

### Beweis zu Beispiel 3.10 - Potenzberechnung

Beweis mittels Berechnungsinduktion. Eigenschaft, die nachgewiesen werden soll:

$$P((x,n),y) :\Leftrightarrow x^n = y$$

d.h. wir wollen zeigen, dass  $h(x,n) = x^n$  gilt.

- ▶ **Induktionsbasis** (Argumente führen zu keinem rekursiven Aufruf)

**Fall  $n = 0$ :**

Ergebnis:  $h(x, n) = 1 = x^0$ , d.h. Eigenschaft erfüllt

- ▶ **Induktionsschritte** (Argumente führen zu rekursiven Aufrufen):

Wir können als Induktionshypothese für die rekursiven Aufrufe verwenden, dass  $h(z_i, k_i) = z_i^{k_i}$  gilt.

**Fall  $n > 0$ ,  $n$  gerade:**

Ergebnis laut Programmcode:  $h(x,n) = y*y$ , wobei  $y = h(x, n/2)$

Wir verwenden die Induktionshypothese: es gilt  $h(x, n/2) = x^{n/2}$

somit  $h(x,n) = y*y = x^{n/2} * x^{n/2} = x^n$ ,

d.h. Eigenschaft in diesem Fall auch für  $n$  erfüllt.

**Fall  $n > 0$ ,  $n$  ungerade:**

Ergebnis laut Programmcode:  $h(x,n) = x * h(x,n-1)$

Wir verwenden die Induktionshypothese: es gilt  $h(x, n-1) = x^{n-1}$

Somit  $h(x,n) = x * h(x,n-1) = x * x^{n-1} = x^n$ ,

d.h. Eigenschaft in diesem Fall auch für  $n$  erfüllt

- ▶ **Induktionsschluss:** somit folgt für alle  $n \geq 0$ , dass  $h(x,n) = x^n$  gilt.

Oft ist es etwas umständlich und aufwendig, so einen Induktionsbeweis formal exakt aufzuschreiben. Aber in den meisten Fällen wird ein Softwareentwickler das auch gar nicht ganz formal durchführen, sondern nur die Denkweise übernehmen. Um zu überlegen, ob meine rekursive Methode richtig funktioniert, muss ich als Softwareentwickler „nur“ folgendes überlegen:

1. Funktioniert die Methode richtig in den Fällen, die nicht zu rekursiven Aufrufen führen?
2. Funktioniert die Methode richtig in den Fällen, die zu rekursiven Aufrufen führen? Um mir das zu überlegen, kann ich davon ausgehen, dass die rekursiven Aufrufe korrekt funktionieren.

Nun sollten Sie das selbst üben:

### *Aufgabe 3.12 - Berechnungsinduktion*

Beweisen Sie mittels Berechnungsinduktion, dass folgende Methode  $g$  für Parameter  $n \geq 0$  das Quadrat von  $n$  berechnet, d.h.  $g(n) = n^2$ .

```
public static int g(int n) {  
    if (n == 0 || n == 1) {  
        return n;  
    } else if (n%2 == 0) {  
        return 4 * g(n/2);  
    } else {  
        int k = g(n+1);  
        return k - 2*n - 1;  
    }  
}
```

## Fazit zu Lektion 3

---

### *Das sollten Sie in dieser Lektion gelernt haben*

- ▶ Mit welchem Lösungsansatz kommt man zu rekursiven Problemlösungen?
- ▶ Was ist der Unterschied zwischen linearer Rekursion und Baumrekursion?
- ▶ Wie kann man durch Berechnungsinduktion nachweisen, dass eine rekursive Methode korrekt ist?